Outsourcing Access Control for a Dynamic Access Configuration of IoT Services

Philipp Montesano, Marc Hüffmeyer and Ulf Schreier
Hochschule Furtwangen University, Robert-Gerwig-Platz 1, Furtwangen im Schwarzwald, Germany

Keywords: Internet of Things, Attribute based Access Control, Safety and Security, Access Control as a Service, REST.

Abstract: The paper describes a lightweight mechanism for authorizing access to IoT resources within distributed systems. As more and more IoT devices arise, the demand for privacy and security increases. But since current solutions are developed for conventional devices, the paper pursues the target of simplifying and applying approved technologies, such as OAuth, to meet special requirements of IoT devices. Therefore, the implemented architecture follows the idea of sourcing the access control logic out, simplifying the logic of the IoT device. Furthermore, the great diversity and fast change of IoT environments is supported by flexible policies and a dynamic and scalable access control system. Performance tests show that sourcing the access control logic out also helps to reduce the amount of consumed memory on an IoT device, in case that complex access logic is given.

1 INTRODUCTION

Though the *Internet of Things* (IoT) is an emerging market, its popularity and chance of success is highly associated with its safety and security. The added value in the fields of assisted living (e.g. *Smart Home*), sustainability (e.g. *Smart City*) and cost-efficiency (e.g. *Industry 4.0*) contributes to IoT being a trendsetting technology. Due to private, public and economic actors as key drivers, IoT services are characterized by a large heterogeneity and a growing number which makes security concerns even more serious (Zhang et al., 2014). Since the internet is still the core of IoT, old security issues become more important while new risks arise: Next to data security and privacy, the physical safety of machines and human operators is a new aspect which need to be considered when implementing IoT systems. But IT security models are mostly designed for conventional systems with high computing power and large hard disk capacities, whereas IoT devices only provide limited resources (cf. Section 2.1) in terms of CPU performance, memory capacity and energy supply (Jing et al., 2014). Further on, these limitations must be involved in the process of implementing a secure environment for IoT devices: As well as IoT itself, security models must be scalable, manageable, lightweight and must perform properly in distributed systems (Gusmeroli et al., 2012).

This work therefore focuses on the special security subtask of access control, i.e. who can access which IoT services and data under what conditions. It investigates the question of how approved techniques can be simplified and applied to an IoT environment, providing secure access to IoT resources while meeting specific needs of IoT devices. Furthermore, it needs to be analyzed if the great diversity and fast pace of IoT environments can be covered by flexible policies and their dynamic configuration. The goal of our work is to outsource access control logic using widely established technologies such as OAuth. By outsourcing the access control application logic to a more powerful network participant, the actual device will be relieved. Outsourcing the access control logic to a centralized component must be done with respect to the constraints, given in the IoT context and the challenges that come up in distributed systems. In addition to the relief of the IoT device, the centralization of the access control system moreover enables a redundancy-free and consistent management of the access policies by which no synchronization of access control information between IoT devices is needed. The main focus of attention is the authorization on the basis of *Attribute Based Access Control* (ABAC) together with the employment of a *User-to-Token* approach (Sandhu and Samarati, 1996) to minimize authentication and authorization efforts at the IoT device. Thus, an access control system with a

Montesano, P., Hueffmeyer, M. and Schreier, U.
Outsourcing Access Control for a Dynamic Access Configuration of IoT Services.
DOI: 10.5220/0006257000590069
In *Proceedings of the 2nd International Conference on Internet of Things, Big Data and Security (IoTBDS 2017)*, pages 59-69
ISBN: 978-989-758-245-5

specific policy language is implemented, providing a well performing authorization structure to outsource access control logic. Furthermore, this approach enables a dynamic configuration of IoT devices to user privileges. It is easy to express who might access which devices under what conditions and to change this configuration.

One application scenario for such a system is a smart home, where heating, lights or garage doors can be controlled remotely and the front door locks/unlocks automatically once dedicated devices enter or leave a range of 1m. Heating and lights might be (remotely) controlled either by the residents or anyone who is actually inside the house. The front door and the garage door might be controlled from dedicated devices, e.g. dedicated smart phones. Therefore, different policies are required that check different conditions like *is a resident* or *is inside the house*.

This work first provides a basic understanding of the applied technologies and procedures (Section 2). This leads to the conceptual architecture (Section 3), whereupon a basic proof of concept is implemented and evaluated (Section 4). Further on, the paper refers to related work (Section 5). Finally, we draw some conclusions and indicate future work (Section 6).

2 FOUNDATIONS

This section describes the foundations of this work, introducing key characteristics of IoT devices and existing technologies upon which the architecture builds.

2.1 IoT Devices

IoT describes the networked integration and interconnection of formerly unconventional everyday objects with embedded systems into the internet. After making the devices communicative by IP addressing, the physical objects are now represented as *Cyber-Physical Systems* (CPS) in a network. As a consequence and due to their sensors and actuators, the devices are capable of monitoring, operating, regulating, optimizing and mining data. But as already stated in Section 1, IoT devices lack of some characteristics, compared to traditional technical appliances (Gusmeroli et al., 2012; Jing et al., 2014):

Processing Performance and Memory Capacity. Due to cost efficiency, devices are highly oriented according to their target applications and needs, which is why simple devices may only provide small processors with low computing power. The same applies to the availability or capacity of memory.

Energy Supply. Moving devices are usually powered by batteries: Hence, their power supply is limited. This is why IoT devices must consume as little energy as possible.

Bandwidth. A low connectivity can be a result of operating in outdoor environments, requiring wireless communication.

Updatability and Connectivity. Being distributed, updating IoT devices or whole environments can be tough, once the system is in live operation. Further on, IoT devices must be accessible by various other devices and applications.

These limitations cause consequences in design requirements for IoT applications:

Distributed Architecture. Outsourcing application logic to powerful hosts reduces the processing efforts of the actual IoT devices. *Representational State Transfer* (REST) (Fielding, 2000) is an architectural style to describe highly scalable and well performing, distributed systems. Therefore, REST is an ideal candidate to build IoT applications on. Still, the communication has to be reduced, using lightweight protocols for instance.

Customized Scope of Application. The highly tailored integration of software and hardware often restricts IoT devices to very specific functions. Therefore, the main focus of its software and processing capacity lies on the core application which allows additional logic (such as access control) to be outsourced.

Real-Time Ability. Availability and fast response times are frequently demanded. On the application layer, a lean software with as little processing time as possible supports the requirement of quick responses.

Protocols. Since HTTP supports REST and is one of the most widespread royalty-free protocols, it brings together as many devices and applications as possible.

After an awareness of the limitations of IoT devices has been created, it is clear that IoT environments need adjusted architectures to fulfill their requirements.

2.2 OAuth

OAuth is an open protocol, designed to share user data (resources) between HTTP clients and server applications without revealing user credentials (Boyd, 2012; Sun and Beznosov, 2012). Version 2.0 of the protocol is defined by the Internet Engineering Task Force in RFC 6749[1]. The protocol introduces four major roles:

Resource Owner. The subject that is capable to grant or prohibit access requests to dedicated re-

[1] https://tools.ietf.org/html/rfc6749

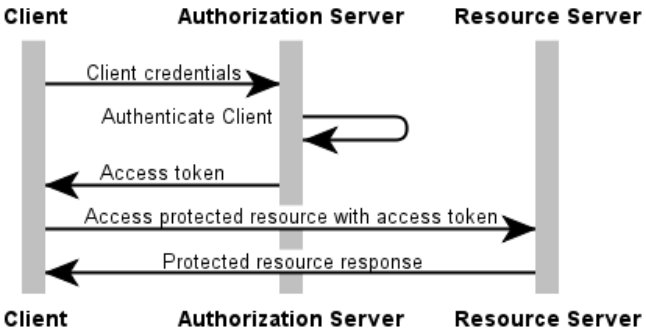

Figure 1: OAuth - Client Credential Grant flow.

sources. This role is also referred to as **User** in case that the resource owner is a human.

Resource Server. The aforementioned resources are located at a resource server. The server only grants access to the resources if the requesting party has a valid access token.

Client. A client is an application that wants to access protected resources of another application.

Authorization Server. A server that assigns access tokens to clients, depending on the access decision of the resource owner.

OAuth also specifies several authorization flows such as the *Authorization Code* flow, also known as *Web Server* flow: A user (a resource owner) utilizes an user agent, such as a browser, to access a client application. The client application wants to access protected resources and therefore starts the OAuth process. A redirect response from the client application to the authorization server is returned back to the user. For example, if the client application wants to access the user's data on Facebook or Twitter, the client application sends a redirect to an authorization server, including the privileges (scopes) that the client application asks for. The user agent performs this redirect and the authorization server responds with an authorization interface. The user might enter its credentials and either grant or prohibit the access request. If the request is granted, the authorization server creates an access code and returns it back to the user agent. The agent in turn forwards it to the client application. The client application can now request an access token from the authorization server using the access code. Once a valid access code is sent to the authorization server, the server responds with an access token that can be used to access the resource. The resource server validates the access token either by asking a token inspection endpoint of the authorization server or by checking the signature of the token. If the token is valid, the server reads the associated scope information of the access token and returns the requested resource to the client. To summarize, an user delegates access rights on his own data from one application to another application. These are ad hoc interactive decisions by a resource owner using a browser (or more generally, an user agent). Access rights are conveyed as scopes which are identifiers with a simple structure. It is the task of the resource server to translate the scope into an access right. Beyond this delegation of access to owned resources, OAuth does not specify any kind of access control model (defining who can access which resources under what conditions).

OAuth's *Client Credential Grant* flow (cf. Figure 1) is a simplified version of the *Authorization Code* flow. In this case, clients are allowed to access resources without asking the user for permission. This case can be used, if the client itself owns the data or if there is a previous arrangement with the authorization server (RFC 6749, section 4.4, p. 39).

2.3 ABAC with RestACL

Due to the fact that IoT devices are integrated in distributed systems, an efficient access control system, which meets the requirements of such environments, is required. The large heterogeneity and a growing number of devices must also be supported, as well as an increasing complexity of access rules. In contrast to traditional access control models, *Attribute Based Access Control* (ABAC) seems to be an appropriate concept that can handle these needs (Ferraiolo et al., 2015). *Role Based Access Control* (RBAC), for example, builds on thoughtful role engineering with which users are assigned to certain roles. Permission is associated with these roles, no matter which other conditions exist. If only few roles exist, RBAC can be the model of choice, but with a growing variety, RBAC reaches its scalability limit as you cannot define roles for every single access condition without facing explosions in the number of roles (Gusmeroli et al., 2012). The advantages of ABAC show up with the need of complex, various and frequently changing access policies. Its biggest benefit is the flexibil-

ity, as permission can be assigned to entities whose attributes meet the criteria of certain policies: Permission is no longer related to roles, but to attributes. Permission to access other information can depend on additional policies by which externals, such as after-sales services, can monitor the devices.

Due to the higher complexity of evaluation, ABAC needs an efficient access control mechanism. (Hüffmeyer and Schreier, 2016a) introduces the *REST Access Control Language* (RestACL) which is suited for resource oriented environments due to its scalability and utilization of the concepts of Representational State Transfer (REST) (Fielding, 2000). Furthermore, RestACL is built on the paradigms of the ABAC model (Hüffmeyer and Schreier, 2016b).

RestACL uses so called *Domains* to map between resources and access policies. An entry consists of a resource, addressed by a *Uniform Resource Identifier* (URI), and subresources of that resource are mapped to access policies. Listing 1 shows a RestACL domain. The domain contains one resource (the garage door from the application scenario) and `GET` or `PUT` access to this resource is handled by policy `P1`. Sending a `PUT` request to the `state` subresource of the `garage` resource might open or close the physical garage door. Sending a `GET` request reads the actual state which allows to check whether the door is opened or closed. Policy `P1` checks whether the `PUT` request is sent from a dedicated device. For example, a dedicated device might be the smart phone of a single resident.

```
{
  "uri":"https://my.smarthome.com",
  "resources":[{
    "path":"/garage/state",
    "access":[{
      "methods":["GET, PUT"],
      "policies":["P1"]
    }]
  }]
}
```

Listing 1: A RestACL domain protecting one resource.

During the winter break, the residents now might go to holiday. While the residents are on vacation, the neighbors might be allowed to park inside the garage. Therefore, a new policy `P2` can be created that checks if the requesting subject is the device of the neighbor (using a first attribute like a `device code`) and if the date is during the holidays of the residents (using a second attribute like `date/time`). This policy can be simply added to the policies that control access to the garage door state to enable the neighbors to open or close the garage door.

Due to space limits, Listing 2 shows a simpler RestACL policy which checks against the attributes of an access control request: According to the `function` and its `arguments`, an `effect` is returned, if the `function` computes to true. To access the garage `state` resource mentioned in Listing 1, the value of the attribute (the `device code` to identify dedicated devices) must be equal to `123456789`. If this applies, the access request is permitted, which means that the requesting subject is allowed to access the resource. The implemented prototype therefore takes advantage of RestACL as its security mechanism along with ABAC as its security model.

```
{
  "policies":[{
    "id":"P1",
    "effect":"permit",
    "priority":"1",
    "condition":{
      "function":"equal",
      "arguments":[{
        "category":"device",
        "designator":"code"
      },{
        "value":"123456789"
      }]
    }
  }]
}
```

Listing 2: A RestACL policy depending on the `device code` attribute.

Listing 3 shows an access control request using RestACL. The above mentioned domain entry is used to map such a request to a policy (cf. Listing 1). The attributes of the access control request are then evaluated against the access conditions, specified in the policy. Resource orientation matches both the distributed system and the access control logic and therefore enables a fast mapping of security policies, based on the resource and the access method. Domains and policies are held in different *Repositories* which enables a simplified and independent permission management.

```
{
  "uri":"https://my.smarthome.com/garage/state",
  "method":"PUT",
  "attributes":[{
    "category":"device",
    "designator":"code",
    "value":"123456789"
  }]
}
```

Listing 3: A RestACL access control request containing the `device code` attribute.

3 ARCHITECTURE

The architecture benefits from the advantages of ABAC and regards the limitations of IoT devices: Outsourcing the access control system simplifies the actual application logic, but raises the communication effort. The centralized repositories as shared resources furthermore implicate benefits, such as non-redundancy, changeability and data consistency. The system can simply be extended, as only the domains and policies of new resources have to be added to the central repositories. Thus, it allows the integration and modification of both resources in the environment and protection rules in the access control system with little effort. Due to its architecture, the system is an example for following the principles of REST.

3.1 Design Guidelines

Outsourcing access control logic is not a trivial task and the architecture of the distributed access control system should respect the following design guidelines.

Reduced Device Efforts. The CPU and memory load of the IoT devices are to be reduced. Therefore, an easy manageable, centralized access control system with a dynamic configuration of resource permission is required.

Simplified Client Authentication. All clients have to authenticate themselves to the IoT device, except trusted clients. Therefore, a token approach is used to reduce authentication efforts at the IoT device side. A more powerful *Authorization and Access Control Server* carries the authentication efforts and issues *Tokens* that can be easily validated at the device side. Therefore, no complex authentication procedure is necessary between clients and IoT devices. Hence, even the trust mechanism is relocated from the IoT device to the authorization and access control server.

User Authentication. Besides authentication of client applications, authentication of users has to be considered when access conditions depend on user attributes. If the client is trusted by the server, the client might handle the user authentication itself and simply include user attributes in the token request. Otherwise, more complex solutions could redirect the user authentication to the authorization and access control server using for instance the OAuth Authorization Code flow. This scenario would enhance the system towards a complete identity and access management. Since the research questions of this paper remain in both cases the same, the implemented solution assumes the simpler case of trusted clients although the architecture supports both cases.

The following architecture assumes that user attributes are part of the RestACL access control request (cf. Listing 3) and that the client can be trusted directly by the authorization and access control server. Mechanisms to establish trust in the triangle user, access control and IoT are not in the scope of this paper.

Configuration Regulations. All IoT devices must be registered at an authorization and access control server. The authorization and access control server information is deposited whilst the setup of the IoT device by a policy engineer. Note that this might be automated, depending on the application. The configuration includes setting of the token lifetime and of policy preferences.

Scalability. Several authorization and access control servers must be capable to manage a variety of IoT devices. As further authorization and access control servers and especially IoT devices can be integrated dynamically, the architecture must be scalable and highly flexible to support various kinds of application and access logic.

Flexibility. Various types of access conditions must be representable. Therefore, permissions are only granted on the basis of attributes. The authorization and access control server checks access rights of all clients according to the attributes. The attributes must comply with the semantics of the access policies.

Privacy. The transmission protocol is generally universal. Nevertheless, a secure (and private) transmission must be ensured (Section 4.1).

Reduced Communication Overhead. As the least possible communication effort is intended, the number of exchanged messages must be reduced to a minimum. Hence, the IoT device should not consult the authorization and access control server after receiving a token.

3.2 Components and Communication Flow

Sharing data with multiple users is a crucial requirement in IoT environments. Because an IoT device might provide several resources, it might not be sufficient to ask for user permission every time a resource is requested. Therefore, we utilize an attribute-based access control system, that enables the user to specify various kinds of access policies at the authorization and access control server side. In addition, we used the concept of *Client Credential Grant* flow (with slight modifications of the request and response formats) as token flow sequence between clients, authorization and access control server and IoT devices. Using this combination enables a much more dynamic

access control system that is founded on established technologies. If a client requests access to a resource of an IoT device, the authorization and access control server checks if the access policies for this resource are met before granting an access token.

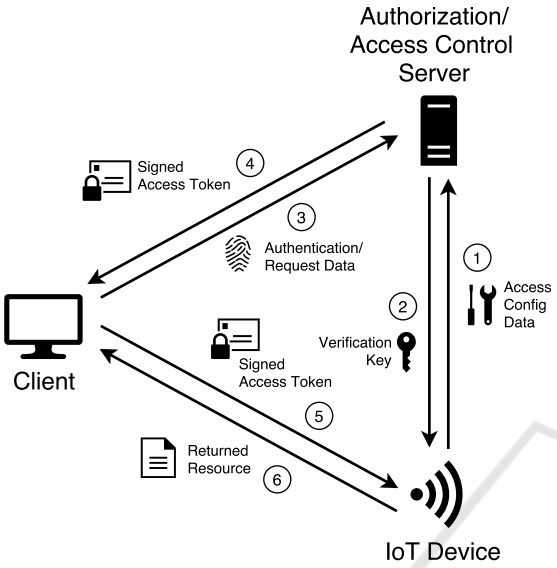

Figure 2: Distributed Access Control Architecture.

Figure 2 depicts the components and the main communication flow. Once an IoT device is set up and the URI of the authorization and access control server is configured, the device starts its one-time configuration process. Therefore, the device sends configuration data to the authorization and access control server (Message 1). These data contain the token lifetime as well as the access policies that must be evaluated, in case the resources of the device are tried to be accessed. If the configuration data are valid, the authorization and access control server responds with a verification key that can be used to validate access tokens that are issued by this authorization and access control server (Message 2). From now on, the IoT device does not need to contact the authorization and access control server again. Note that the devices still might send changes of the access configuration to the authorization and access control server if desired. Resources are directly addressable, using their unique URI. If a client requests a resource of the IoT device, a token (including a signature) is expected and must be successfully validated. Otherwise the IoT device must not grant access to the resource. The IoT device therefore compares the actual request to the request conditions, given by the token (e.g. the device compares whether the source IP address of the incoming request matches the client IP address mentioned in the token). If the request matches the token and the validity period has not already been expired, access can basically be granted.

A client now may authenticate itself to the authorization and access control server and ask for a token (Message 3). Therefore, the client sends the user authentication data as well as request data (the URI of the desired IoT resource, the access method and, in case that the client is trusted, the subject attributes, which are supplied by the client). Further attributes, e.g. attributes of the requested resource or environmental attributes, like the actual date, are allocated at the authorization and access control server. If the authentication procedure was successful and the policy evaluation indicates that the client is allowed to access the resources, the authorization and access control server issues a token and returns it back to the client (Message 4). The implemented prototype uses special HTTP status codes to indicate whether a token could be issued. If a token was issued, the authorization and access control server responds with the code 280 and returns the token. If the token could not be issued, the server responds with the status code 480. Note that Figure 2 shows a simplified version of the communication flow to increase the readability. Clients likely do not know the related authorization and access control server, if they try to access a resource of an IoT device for the first time. Therefore, a client usually tries to access the resources of an IoT device without a token. The IoT device then refuses this request and responds with a redirect that points to its related authorization and access control server. From now on the client might ask the authorization and access control server for an access token before trying to access the resource of the IoT device.

Finally, the client contacts the IoT device, transmitting the issued token (Message 5). If the token matches the request and the signature is valid, the resource is returned (Message 6). The implemented prototype does so by using special HTTP status codes: 290 for a successfully validated request, 490 in case that the validation has failed.

For example, the resident of the smart home might want to open his garage door. The resident therefore sends a request to the authorization and access control server containing his authentication data. The authorization and access control server loads various attributes of the requesting subject, the requested resource and environmental attributes like the actual time and date. In the application scenario, for example, the authorization and access control server loads at least the `device code` attribute of the requesting device. The authorization and access control server therefore either must implement its own attribute repository or employ external attribute sources

such as identity providers to load the attributes. Details about attribute provisioning can be found in (Hüffmeyer and Schreier, 2016c). Note that clients might also send attributes in case that the client can be trusted. The authorization server formulates an access request and passes the access request to a RestACL engine. The engine then checks which policies need to be evaluated in order to access the garage state resource. These policies are then evaluated against the attributes. If the evaluation was successful, the authorization and access control server delivers a signed access token back to the resident. Using this token, the resident now can open the garage door.

3.3 Token Concept and Integrity

Tokens provide the link between authorization and access control servers and IoT devices and replace their direct communication. Although clients need to ask the authorization and access control server for a token, they are not generally valid, but issued for dedicated requests. If the attributes of a user match the access policy of the authorization and access control server, the client is given a refined and specialized OAuth like token. Considering Listing 4, tokens contain the scope of the request (or a set of requests). Therefore, a token has a validity period (resp. a expiration time stamp) and contains all IP addresses of all requesting subjects. Since the authorization and access control server only authorizes explicit requests and returns customized tokens, the IoT device simply has to verify the terms of the token against the actual request (see Section 4.1 for thoughts on *Replay Attacks*). If the token matches the request, access is granted. Due to a signature, tokens cannot be modified during their transference.

```
{
  "token":[{
    "lifetime":"01.01.2017 12:01:00",
    "resources":["https://my.smarthome.com/
                                garage/state"],
    "device":"https://my.smarthome.com/garage",
    "client":"1.2.3.4",
    "auth_server":"https://my.smarthome.com/
                                authorization",
    "method":"GET"
  }], "signature": [{
    "value":"1a7b6062c6fb8fc2511036df6178 ..."
  }]
}
```

Listing 4: A sample token as issued by the authorization and access control server.

Every token is signed after its creation with a hash value. To ensure integrity, the hash value is encrypted by the authorization and access control server with a private key. The authorization and access control server provides the corresponding public key (the verification key) to IoT devices during its configuration process. Knowing the public key, the IoT device can decrypt the signature and compare it with its own hash computation to validate the integrity. To verify the integrity of the token, the IoT device hashes the token itself, decrypts the signature of the token and compares both hash values. If they match, the token is valid and the resource is returned to the client – if they differ, the token has been modified during its transmission (Section 4.1) and an error code is returned. Although every participant can possibly decrypt the signature (when knowing the public key), none of it can encrypt a modified hash value, since the private key is merely known by the authorization and access control server. Furthermore, only authenticated clients can apply for a token at the authorization server. Hence, no more authentication against the IoT device is required.

4 PROOF OF CONCEPT

The proof of concept focuses on centralized access control in distributed systems combined with token concepts. It therefore only provides one component in a whole security environment, as stated in Section 4.1. In return, it takes a closer look at the integration of a particular access control system. Further on, the concept realizes the setting of the token lifetime, an entry in the domain repository and the applied access polices through manual input at the IoT device. The system relies on HTTP using proprietary status codes. It makes use of HTTP, also because of its benefits, such as the native support for mobile devices, the convenient usage within `Jersey`[2] and the free use, due to it being an open standard. It can also be replaced by HTTPS subsequently. Further on, all of the data is transmitted within header fields.

4.1 Security and Privacy Considerations

As the concept offers a security mechanism on the application layer, it can rely on additional security mechanisms such as *Transport Layer Security* (TLS) and authentication procedures. As the proof of concept focuses on the interaction of RestACL, tokens and IoT devices, it provides integrity of a token due its signature (encrypted hash) in a first version. In a second version *End-to-End Encryption* (E2EE) is used to fully encrypt the token. This ensures privacy and

[2]https://jersey.java.net

prevents identity theft or *Man-in-the-Middle Attacks*. To enable E2EE, a symmetric key is exchanged between the authorization and access control server and every IoT device during its configuration phase (after the public key has been received): This *PreShared Key* (PSK) cannot be spied out due to its asymmetric encryption. In addition, replay attacks can be avoided by including a unique usable *Nonce* to ensure that every request is fresh.

4.2 Experimental Results

The implementation uses a *Raspberry Pi 3 Model B* as its IoT device: It features a quad-core *System on a Chip* (SoC) ARM processor with 1.2 GHz CPU speed, 1 GB RAM and a (wireless) network interface. It offers 4 resources, mapped to 3 physical components: A temperature sensor, which constantly writes the temperature into a file, can be accessed with a *GET* request. Next, a LED diode can be switched on briefly, using a *PUT* request. *POST* and *DELETE* requests are represented by a LCD module, which the client can write on or clear. Further on, the authorization and access control server and the client are represented by a 3.2 GHz iMac (4 GB RAM) and a 2 GHz MacBook (8 GB RAM) within one wireless network. Note that the 1.2 GHz Raspberry Pi 3 is still a very powerful device and considerably longer runtimes are expected when running on smaller devices (such as the *ESP8266*[3]).

The scalability depends on a dynamic integration of entities, policies and resources into the environment and into the access control mechanism. Since an authorization and access control server must handle several IoT devices, which can have multiple resources, RestACL must evaluate quickly, even when the sizes of the repositories rise. Table 1 lists the average runtime for an access control request and its evaluation with the help of RestACL (running on the 2 GHz MacBook) according to the amount of policies and domains. Table 2 shows the total memory consumption is shown. As the experimental results of (Hüffmeyer and Schreier, 2016a) already prove, the runtime remains at a constant level although the amount of domains and policies rises and the memory consumption increase.

To meet further needs of IoT devices, the processing power and memory consumption has to be reduced. It is therefore to question if an own access control mechanism on the side of every IoT device performs better than the presented architecture. Table 3 shows the result of a comparison between the three different security mechanism implementations, all running on the Raspberry Pi: V_1 lists the runtime when an IoT device implements its own RestACL mechanism. Note that there will be additional efforts, related to the authentication procedure which was not included in the test series. In V_1, these efforts have to be provided by the IoT device additionally. V_2 lists the performance of the asymmetrically encoded signature (based on RSA and SHA-512), comparing a plain token to a request, generating an own signature and comparing both signatures (as implemented in Section 4). V_3 represents end-to-end encryption of the whole token and therefore uses symmetric token decryption (based on AES and PSK) and compares the token to a request.

According to the test series, the required CPU processing power can be relieved best with a symmetrically encrypted token (V_3) since its efforts are close to the efforts of the on-board RestACL system (V_1) that lacks the authentication procedure. Again, we expect these results to differ if less powerful IoT devices are employed. In terms of memory consumption, the best solution depends on how many resources and policies must be supported. For larger amounts of resources or policies V_2 and V_3 show clearly lower values while for small amounts of resources and policies V_1 shows slightly better values. It can also be found that RestACL is much more powerful when running on a conventional device than running on a SoC (comparing the runtime in Table 1 and Table 3). The same can be expected for an authentication procedure. The most remarkable thing that one can see from Table 2 is that the On-Board solution shows variable memory consumption depending on how many access policies must be supported. In contrast, the outsourced solution shows fixed (and therefore predictable) values. The required memory only depends on the implementation of the encryption library.

Next to scalability and single processing times, the overall runtime of a request (from the very first request of the client to the last received response) is a further key question. Adding the propagation delay t_{Pd} (nodal processing and queueing) and the network transmission delay t_{Nw} to the runtime t_{Vx} of one of the access control procedures (V_1, V_2 or V_3), the overall runtime t_{TotalV_x} is as follows:

$$t_{TotalV_x} = t_{Pd} + t_{Nw} + t_{V_x}$$

Therefore, t_{TotalV_1} can be computed as:

$$t_{TotalV_1} = t_{Pd_C} + t_{Pd_{IoT}} + 2 * t_{Nw_{C-IoT}} + t_{V_1}$$

with t_{Pd_C} being the propagation delay on the client side, $t_{Pd_{IoT}}$ being the propagation delay of the IoT device and $t_{Nw_{C-IoT}}$ being the network runtime between client and IoT device.

[3]https://espressif.com/products/hardware/esp8266ex

Table 1: Evaluation runtimes.

Policies \ Domains	10	100	1.000	10.000
10	0.14 ms	0.14 ms	0.15 ms	0.15 ms
100	0.14 ms	0.14 ms	0.15 ms	0.15 ms
1.000	0.14 ms	0.14 ms	0.15 ms	0.16 ms
10.000	0.14 ms	0.14 ms	0.15 ms	0.16 ms

Table 2: Memory consumption.

Policies \ Domains	10	100	1.000	10.000
10	1.25 MB	1.32 MB	1.92 MB	7.93 MB
100	1.32 MB	1.38 MB	1.97 MB	7.99 MB
1.000	1.87 MB	1.93 MB	2.52 MB	8.09 MB
10.000	7.35 MB	7.41 MB	8.0 MB	14.02 MB

Table 3: Security mechanism implementations.

Procedure	Runtime	Memory
V_1: On-board	4.98 ms	cf. Table 2
V_2: Asymmetric	34.64 ms	1.7 MB
V_3: Symmetric	5.22 ms	1.7 MB

Regarding V_2 and V_3, t_{TotalV_x} depends on whether the client application performs an initial access request to the IoT device without knowing the authorization and access control server and therefore without a token. t_{Pd_C}, $t_{Pd_{IoT}}$ and $t_{Nw_{C-IoT}}$ need to be considered twice in the very first request if the client application is capable to store the authorization and access control server address for each IoT device (cf. Section 3). If the client application is not capable to do so, they need to be considered twice for every access request.

$$t_{TotalV_{2/3}} = t_{C-AS} + t_{C-IoT} + t_{V_{2/3}}$$

with

$$t_{C-AS} = t_{Pd_C} + t_{Pd_{AS}} + t_{Nw_{C-AS}}$$
$$t_{C-IoT} = t_{Pd_C} + t_{Pd_{IoT}} + t_{Nw_{C-IoT}}$$

Because the propagation delay and network runtimes depend on various conditions, no absolute test series is adequate. But in general one can say that network runtimes are usually in the dimension of hundreds of milliseconds. Taking the long overall runtime into consideration, the decision of sourcing access control out becomes rather a question if a centralized, manageable access control system is required. Notably faster runtimes can not be expected, but the memory consumption can be reduced to a fixed and therefore predictable value on the IoT device. In case that multiple resources are provided by the IoT device and those resources have multiple individual access policies, the memory consumption can be remarkably reduced. Consideration should also be given to the fact that, if authentication was implemented, the client would have to authenticate against the IoT device, leading to a longer runtime of V_1 and to a higher memory consumption of the IoT device. Besides the performance differences, the major benefit of an outsourced access control system is the simplification of policy management, that allows to apply, create, change and remove access permission on multiple devices in a centralized fashion.

5 RELATED WORK

User Managed Access (UMA) is based on the *OAuth* protocol and thus also offers authorization. It focuses on keeping user login data secret, while allowing third parties access to protected resources. It therefore simply passes on *Credentials* instead of unveiling the actual access information (Machulak et al., 2010). UMA generates more communication overhead, as all clients have to be registered in advance and on account of further requests between the resource and the authorization and access control server. Considering this, the benefits of the introduced architecture, such as a lower complexity and less communication overhead, are explained. However, UMA does neither recommend or specify an explicit access control mechanism, nor a policy language, nor the process of validating a token. Therefore, the example architecture is a streamlined instance of an UMA implementation with RestACL as its access control mechanism, creating less communication effort and being specified for the demands of IoT devices. It focuses on the part of authorization, which is why the authorization structure is significantly more developed than

specified in the core protocol of UMA. Unlike UMA, the approach of this paper presupposes requirements which allow a simplification of the UMA protocol.

Problems with outsourcing and sharing data are discussed in (di Vimercati et al., 2007). De Capitani di Vimercati et al. challenge the problem of sharing data in distributed systems without providing it to the public by using selective encryption. Therefore, shared resources are encrypted using a key. Tokens are derived from that key and can be used to access the shared resource. The approach focuses on encryption and the derivation of keys. Access methods are always expected to be read only and an authorization is defined as a double of user and resource whereby users also can be groups.

(Gusmeroli et al., 2012) focuses on a *Capability Based Access Control* (CapBAC), having a token as a representation of the capability to access resources. In contrast to this paper, the initial credential is issued by the owner of the resource, not by an authorization and access control server. Furthermore, no semantics of the access rules are defined.

As Section 4.2 already shows, IoT devices lack of efficient decryption algorithms with which faster processing times could be accomplished. (Zhang et al., 2014) mentions that designing lightweight cryptographic systems is still a challenging task: While a public key system offers data integrity, data privacy and is suited for authentication, it produces more computational overhead.

Constrained Application Protocol (CoAP) is stated as an appropriate protocol for *Machine-to-Machine* (M2M) communication – due to its simplicity and low overhead, it is suited for IoT devices (Raza et al., 2013). Since HTTPS is highly connected to TLS as its security protocol, TCP is required which is processed slower. CoAP utilizes *Datagram Transport Layer Security* (DTLS), running over UDP which is processed faster.

Next to simplifying protocols, (Shafagh and Hithnawi, 2014) focuses its attention on the hardware of IoT devices: Although E2EE only needs the public key crypto system at the configuration process, its memory is allocated during the whole running time of the application – this means less memory for the actual application logic during the entire time span. Hardware encryption engines within recent SoC devices are not only offered for symmetrical encryption, with only little additional cost. Acceleration engines are also offered for public key systems and contribute to faster processing times of encryption tasks.

6 CONCLUSION AND FUTURE WORK

Since SoC devices distinguish from conventional processors, they imply limitations and therefore demand special design requirements. After stating enabling technologies, especially OAuth, ABAC and REST, the overall architecture is revealed. This concludes in a closer look at all involved entities, as well as in demonstrating a proof of concept. Furthermore, the experimental results give an impression of how well RestACL meets the demands of the introduced architecture: Since the runtime remains stable, despite of rising repositories, all domains and policies can be managed by one authorization and access control server. Therefore, it is approved that a centralized access control system with flexible policies can provide a dynamic configuration of IoT entities within a changing environment and therefore offer the possibility of configuring resources dynamically according to attributes. As the introduced Raspberry Pi is still a powerful device, compared to smaller IoT devices, the rising file sizes do not meet the requirements of IoT devices, according to their memory capacity. Especially when implementing authentication, the memory consumption of **V1** increases further. This leads even more to centralizing the access control logic and the trust mechanism, relieving the IoT device. Considering the overall runtime ($t_{TotalVx}$), the decision of sourcing RestACL out depends on practical conditions. Although the runtime increases due to a greater communication effort, the practical benefits of a centralized access control logic are high scalability, ease of use, non-redundancy, changeability, data consistency and more convenient data backups.

REFERENCES

Boyd, R. (2012). *Getting Started with OAuth 2.0*. O'Reilly Media.

di Vimercati, S. D. C., Foresti, S., Jajodia, S., Paraboschi, S., and Samarati, P. (2007). Over-encryption: Management of Access Control Evolution on Outsourced Data. *VLDB '07 Proceedings of the 33rd International Conference on Very Large Data Bases*.

Ferraiolo, D., Kuhn, R., and Hu, V. (2015). Attribute-Based Access Control. In *Computer, Vol. 48*. IEEE Computer Society.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine.

Gusmeroli, S., Piccione, S., and Rotondi, D. (2012). IoT Access Control Issues: a Capability Based Approach. In *2012 Sixth International Conference on Innovative*

Mobile and Internet Services in Ubiquitous Computing.

Hüffmeyer, M. and Schreier, U. (2016a). Analysis of an Access Control System for RESTful Services. *ICWE '16 - International Conference on Web Engineering*.

Hüffmeyer, M. and Schreier, U. (2016b). Formal Comparison of an Attribute Based Access Control Language for RESTful Services with XACML. *SACMAT '16 - Symposium on Access Control Models and Technologies*.

Hüffmeyer, M. and Schreier, U. (2016c). RestACL - An Access Control Language for RESTful Services. In *ABAC '16 Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*.

Jing, Q., Vasilakos, A. V., Wan, J., Lu, J., and Qiu, D. (2014). Security of the Internet of Things: Perspectives and Challenges. *Wireless Networks*.

Machulak, M. P., Maler, E. L., Catalano, D., and van Moorsel, A. (2010). User-Managed Access to Web Resources. In *Proceedings of the 6th ACM Workshop on Digital Identity Management*.

Raza, S., Shafagh, H., Hewage, K., Hummen, R., and Voigt, T. (2013). Lithe: Lightweight Secure CoAP for the Internet of Things. *IEEE Sensors Journal*, 13(10).

Sandhu, R. and Samarati, P. (1996). Authentication, Access Control, and Audit. *ACM Computing Surveys*, 28.

Shafagh, H. and Hithnawi, A. (2014). Poster Abstract: Security Comes First, a Public-key Cryptography Framework for the Internet of Things. In *2014 IEEE International Conference on Distributed Computing in Sensor Systems*.

Sun, S.-T. and Beznosov, K. (2012). The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*.

Zhang, Z.-K., Cho, M. C. Y., Wang, C.-W., Hsu, C.-W., Chen, C.-K., and Shieh, S. (2014). IoT Security: Ongoing Challenges and Research Opportunities. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*.